# Exercises: ASP.NET - Core Introducing Services

Problems for exercises for the ["ASP.NET Core Fundamentals" course @ SoftUni](https://softuni.bg/trainings/4953/asp-net-fundamentals-may-2025)

A movie ticket and popcorn

Description automatically generated

In this workshop, we will restructure our logic to follow best practices. **Instead of accessing the database directly** in our controllers, we will introduce a service layer. This helps us:

* Keep our controllers clean
* **Reuse logic** in multiple places
* Make the application easier to maintain and test

## Create a MovieService Interface

The **interface defines** **what the service should do** — not how. We place service contracts (interfaces) inside the Interfaces folder in the CinemaApp.Services.Core project.

* Inside the **CinemaApp.Services.Core** project, create a new folder called Interfaces
* Inside this folder, add a new interface named IMovieService.cs

A screenshot of a computer

AI-generated content may be incorrect.

A screen shot of a computer

AI-generated content may be incorrect.

### Adding a New Task Method to the Interface

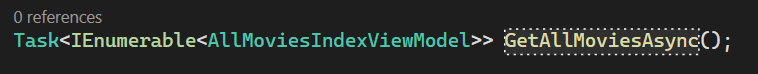
After creating the empty IMovieService interface, we now want to define **what functionality** the service should offer. This step is important because it sets the **contract** between the controller and the service layer.

In our case, we want to **allow the controller to retrieve all movies from the database**, formatted as view models.

* Add the following using at the top of the file:



* Add the following method inside the interface block:



Your interface should now look like this:

A screen shot of a computer program

AI-generated content may be incorrect.

* The interface declares a **Task-returning method** called GetAllAsync()
* It tells the service:   
  *"You must provide logic that returns a list of movies formatted as AllMoviesIndexViewModel"*
* We use Task<> because the operation is asynchronous — it fetches data from a database
* The controller will later **call this method** instead of writing the full logic itself

## Create a MovieService Class

Now that we’ve defined the contract (IMovieService), it’s time to **create the class that will contain the actual logic**. This class will be **responsible for retrieving the movies from the database and preparing them for display**.

This **logic previously lived directly inside the controller**. However, **in applications, we should separate this kind of logic into dedicated services**, keeping controllers clean and focused only on routing and communication with the view.

A screenshot of a computer

AI-generated content may be incorrect.

* The class implements the IMovieService interface, meaning it must define every method declared in it
* Inside the **GetAllAsync()** method:
  + We project the database entities into AllMoviesIndexViewModel

### Injecting the DbContext

A screen shot of a computer

AI-generated content may be incorrect.

* This is called **constructor injection**
* We are telling ASP.NET Core:   
  *“Give me an instance of CinemaAppDbContext whenever this service is created.”*
* CinemaAppDbContext allows the service to query the database

We mark the field as **readonly** because it is **only assigned once**, in the constructor, and never changed afterward.

### Moving the Logic from the Controller

In the previous controller (see below), we had:

A screen shot of a computer program

AI-generated content may be incorrect.

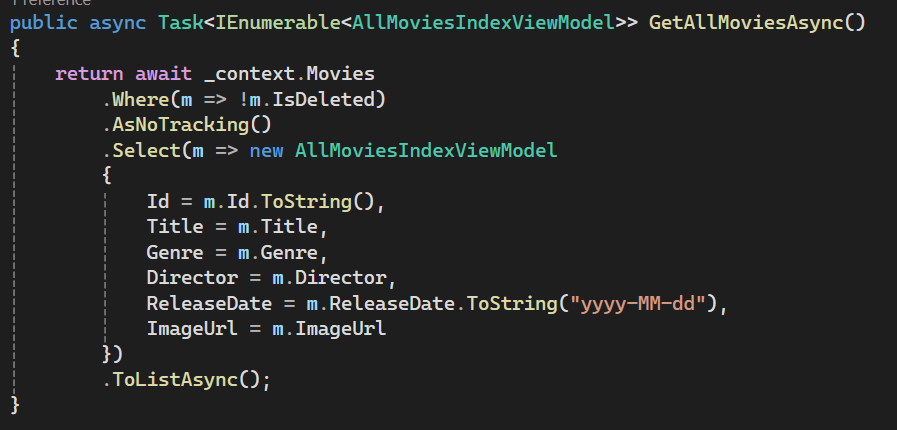
This works, but the controller is now doing **too much** — it retrieves the data, projects it, and prepares it for the view. This makes it harder to test, reuse, and maintain.

To improve the structure, we will **extract this logic into a service**, specifically in the method called GetAllAsync() inside MovieService.

### Creating the Service Method: GetAllAsync()

Now we will define the method that performs the exact same logic — but in the service class instead.

In the MovieService class, **implement** the GetAllAsync() method like this:



* **AsNoTracking()** – improves performance when the results will only be read, not updated.

## Registering the Service in Program.cs

To use the MovieService in our application, we need to **tell ASP.NET Core how to inject it when it is needed** — for example, inside a controller.

**ASP.NET Core has a built-in Dependency Injection (DI) system** that we use for this purpose. We simply register the service and its interface once, and the framework handles everything else.

### Where to Register the Service

Open the Program.cs file from the CinemaApp.Web project.

You will find a section where other services like DbContext and Identity are registered. Add the following line **after** the database context registration:

A computer screen shot of a program code

AI-generated content may be incorrect.

* **IMovieService** is the interface that defines the contract
* **MovieService** is the class that implements that contract
* We use **Scoped** lifetime because we **want a new service instance per HTTP request** (this is the recommended practice for services working with EF Core)

## Refactoring the MovieController to Use the Service

Now that the service is ready and registered, we can **inject it into the controller** and let the service handle the logic of fetching the movie data.

* Replace:

A screen shot of a computer program

AI-generated content may be incorrect.

* With:

A screen shot of a computer program

AI-generated content may be incorrect.

### Update the Index Action

* Replace the old Index() action logic:

A screen shot of a computer

AI-generated content may be incorrect.

* With this clean version:

A black screen with white text

AI-generated content may be incorrect.

### Testing the Application After Refactoring

We have moved the logic from the controller to the service layer and updated the dependency injection, it’s time to **run the application** and make sure everything still works correctly.

A screenshot of a computer

AI-generated content may be incorrect.

## Implementing CRUD Operations for Movie

The MovieService is integrated into the application and working correctly, we can begin building the full set of **CRUD (Create, Read, Update, Delete)** operations for the Movie entity.

### Who Can Add a Movie?

In a real-world cinema application, not every visitor should be allowed to add movies.

For now, we will assume:

* Only logged-in users can add movies to the system.

A screenshot of a computer program

AI-generated content may be incorrect.

* Later in the course (in Workshop III), we will learn how to configure **authentication and authorization** to enforce this rule.
* In this workshop, we will:
  + Display the **"Add Movie"** button in the navigation only for logged-in users
  + **Not** implement role checks or [Authorize] attributes yet

## Implementing the Create Movie Feature

We’ll now build the **functionality that allows logged-in users to add a new movie to the system**.

This will be done using the Create (**GET** and **POST**) actions in the MovieController, but the actual logic will be placed inside the MovieService.

### Create the MovieFormViewModel

We need a **ViewModel** that **matches the form fields used for creating a movie**. It will be used both for displaying the form and processing submitted data.

A screenshot of a computer

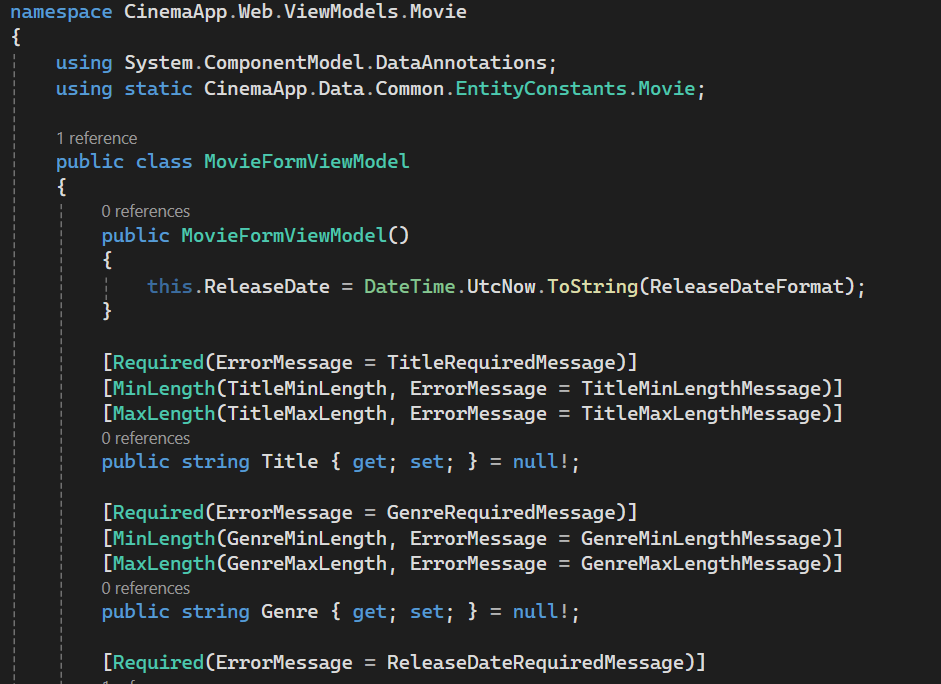
AI-generated content may be incorrect.

A screen shot of a computer

AI-generated content may be incorrect.

### Implement the MovieFormViewModel





### Validation Constants in EntityConstants

To **avoid hardcoding validation rules and error messages directly in the view model**, we define them in a **centralized class called EntityConstants**.



A screenshot of a computer program

AI-generated content may be incorrect.

### Adding the AddAsync Method in MovieService

Now we will implement a method in the MovieService that takes a MovieFormViewModel and stores the data in the database.

This method will be called by the MovieController in the **Create POST action**.

* Update the Interface

A screen shot of a computer program

AI-generated content may be incorrect.

* Implement AddAsync() in MovieService

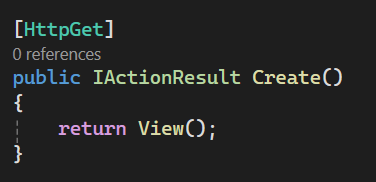
A screen shot of a computer program

AI-generated content may be incorrect.

### Creating the Add Movie Actions in the Controller

Now that we’ve implemented the AddAsync() method in the service, it’s time to expose this functionality to the user by implementing the **Create actions inside the controller**.

* Add the **GET Action**



This action **doesn’t need any data from the service** — it **simply returns the empty form view**.

### Creating the Add Movie Form (Create.cshtml)

Now that we’ve added the Create() GET action, we will create the corresponding Razor view that displays the **Add Movie** form.



A screenshot of a computer

AI-generated content may be incorrect.

A screen shot of a computer program

AI-generated content may be incorrect.

### Test the View

After adding the file, test that the view works correctly:

A screenshot of a computer error

AI-generated content may be incorrect.

### Updating the Navigation Link for Add Movie

By default, the navigation bar in the layout points to an action called Add, which **does not exist**. This will result in a runtime error when clicked.

We need to update it to point to the actual action: **Create**.

A computer screen with text

AI-generated content may be incorrect.

A computer screen with text

AI-generated content may be incorrect.

Once you update the link, the Add Movie option in the navbar will open the form as expected when you're logged in.

A screenshot of a computer

AI-generated content may be incorrect.

### Implementing the Create (POST) Action in the Controller

Once a user fills out the Add Movie form and clicks the **submit** button, the form sends a **POST request to the server**. This **request is handled** by a Create(MovieFormViewModel model) action inside the MovieController.

* Open your MovieController.cs and add the following **POST action**:

A screen shot of a computer code

AI-generated content may be incorrect.

* **Model Binding**: ASP.NET automatically binds the form fields to the MovieFormViewModel parameter.
* **Validation**: The [Required], [MaxLength], [Range], etc., rules defined in the ViewModel are automatically checked. If any rule fails, ModelState.IsValid becomes false.
* **Return to Form on Error**: If the data is invalid, the form is shown again with validation messages.
* **Call to Service**: If the model is valid, we call the AddAsync() method from the MovieService, which handles the logic of saving the movie to the database.
* **Redirect**: After a successful addition, the user is redirected to the movie listing (Index) to see the updated list.

A screenshot of a computer

AI-generated content may be incorrect. A screen shot of a movie

AI-generated content may be incorrect.

You should be redirected to the Movies page — and the new movie should be visible in the list

A white rectangular object with black lines

AI-generated content may be incorrect.

## Implement the Details Page

The **Details page** will allow users to view full information about a specific movie. This includes title, genre, release date, director, duration, description, and image.

### Creating the Details View

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.



A screenshot of a computer program

AI-generated content may be incorrect.

### Planning the ViewModel Based on the View

Before we create the MovieDetailsViewModel, it’s important to understand that **the view determines what the ViewModel should contain**.

From the structure of the Details view, we know we’ll need:

* **Title**, **Genre**, **Release date** (formatted), **Director**, **Duration**, **Description**, **Image URL**

### Create the MovieDetailsViewModel

A screenshot of a computer program

AI-generated content may be incorrect.



A screen shot of a computer program

AI-generated content may be incorrect.

### Injecting the ViewModel into the Razor View

To display data in a Razor view, we must tell the view **what model it expects**. This is done at the top of the .cshtml file using the @model directive.

A screen shot of a computer program

AI-generated content may be incorrect.

### Adding the GetByIdAsync Method in the MovieService

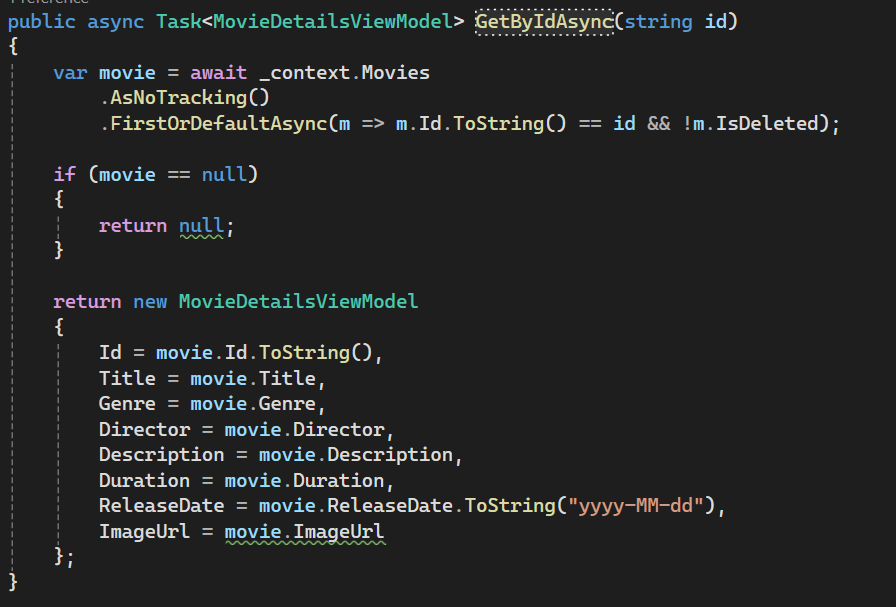
To load the data needed for the Details view, we’ll define a service method that fetches a single movie from the database using its ID.

* Open the interface file and add the following method declaration:

A screen shot of a computer program

AI-generated content may be incorrect.

* Implement the Method in MovieService.cs



### Implementing the Details Action in MovieController

A computer screen shot of a program code

AI-generated content may be incorrect.

* **Receive a movie ID** as a route parameter
* **Call the service** to retrieve the corresponding movie
* **Return a 404 error** **if** the movie doesn’t exist
* Otherwise, **return** the Details.cshtml view with the view model

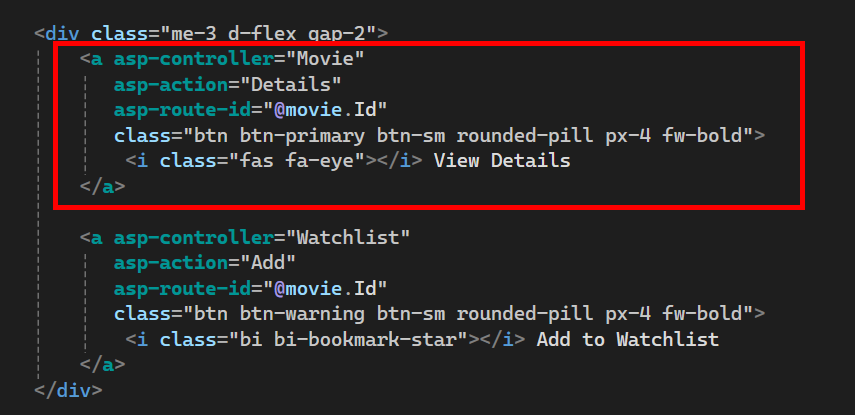
### Replace in Views/Movie/Index.cshtml the View Details Button with a Link

Before (current code):

A computer screen shot of a black screen

AI-generated content may be incorrect.

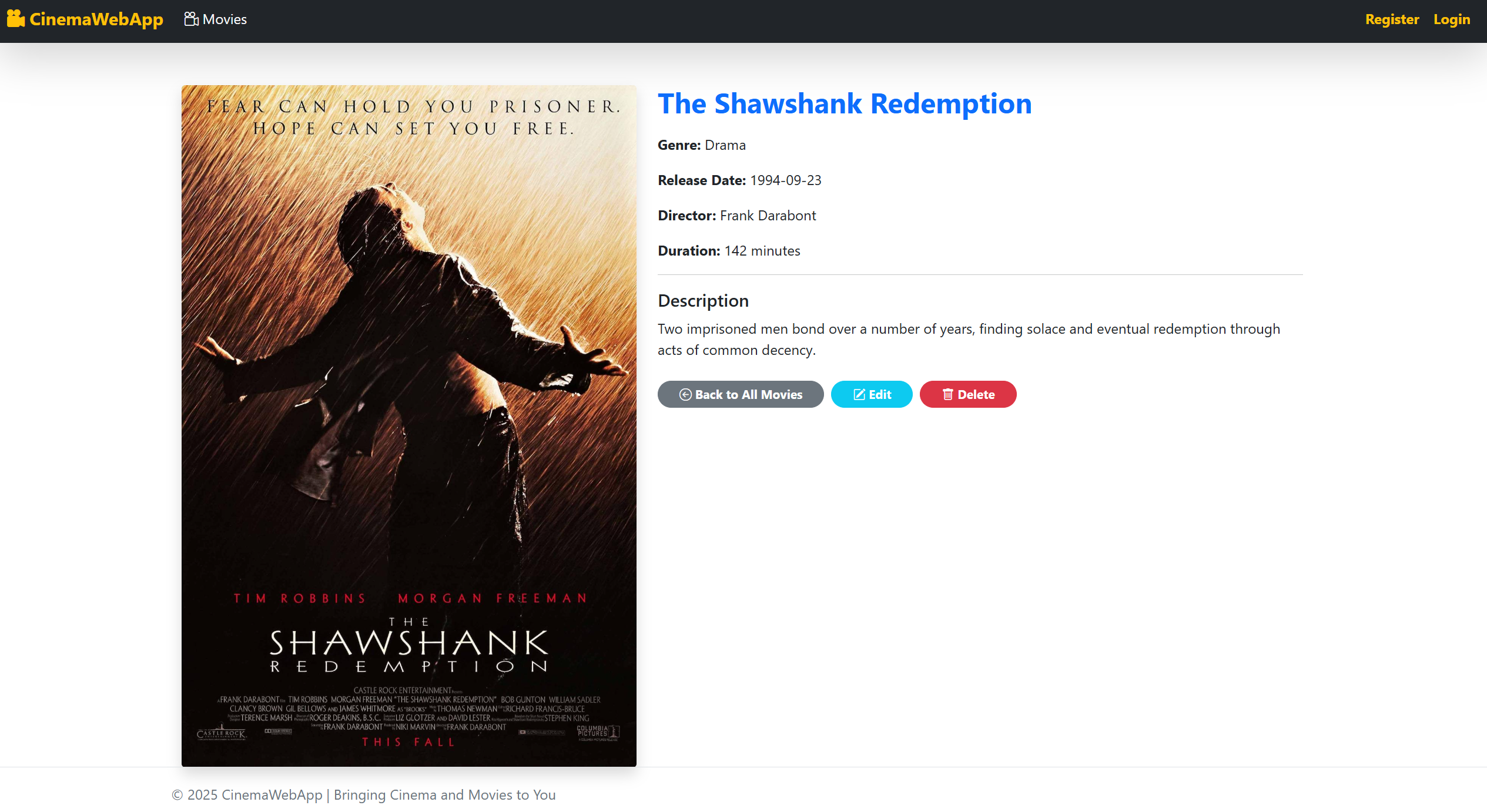
After (updated code):



### Updated Movie Details View (with Edit and Delete Buttons)

Since we will handle authentication and authorization in the **next workshop**, we’ll simplify the view for now by **removing the User?.Identity?.IsAuthenticated condition**. This way, all users can see the Edit, Delete, and Watchlist buttons temporarily.

Here’s the updated section that provides the cleaned-up **Details.cshtml** view.



## Implement the Edit Page

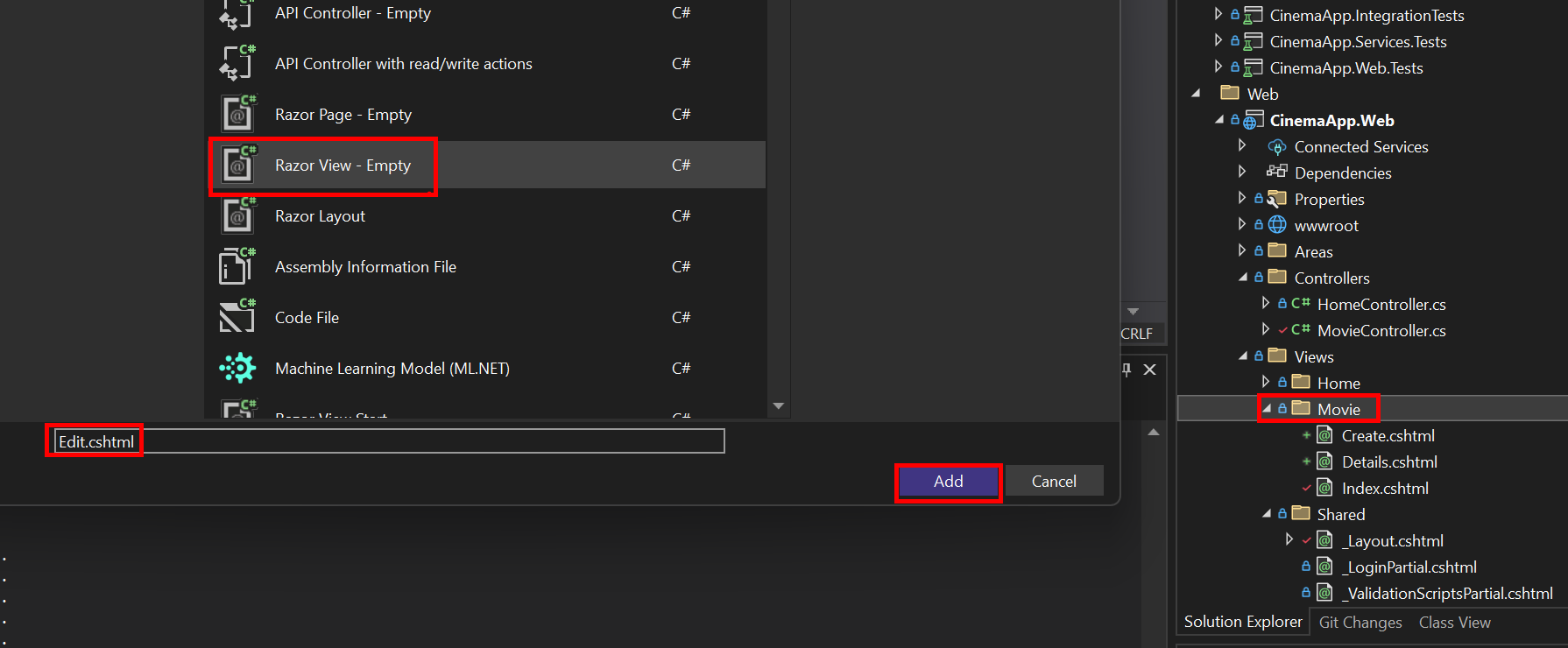
We will now allow to **edit existing movies**. The edit functionality reuses the MovieFormViewModel we created earlier for the Create action.

The process is similar to Create and Details:

* Show the Edit form with existing data (GET)
* Save the changes on form submission (POST)

### Creating the Edit View





A computer screen shot of a program

AI-generated content may be incorrect.

### Add Interface Methods to IMovieService

A screen shot of a computer program

AI-generated content may be incorrect.

### Implement the Methods in MovieService.cs

A screen shot of a computer program

AI-generated content may be incorrect.

A screen shot of a computer

AI-generated content may be incorrect.

### Add Edit Actions in the MovieController

* Add the **GET Edit Action**

A computer screen shot of text

AI-generated content may be incorrect.

* Add the **POST Edit Action**

A screen shot of a computer program

AI-generated content may be incorrect.

* We use the same MovieFormViewModel as for the Create form
* If the ID is invalid or the movie is not found, we **return 404**
* If the input is valid, the **service updates the record in the database**
* The user is **redirected back to the Details view** of the updated movie

### Updating the ViewModel to Include the Movie Id

When implementing Edit functionality, the application needs to know **which movie is being edited**. To achieve this, we must include the movie’s ID in the view model used in the form.

We will **update the MovieFormViewModel to include a new Id property**:

A screen shot of a computer program

AI-generated content may be incorrect.

* We’ve added the Id property to the MovieFormViewModel, the next step is to **populate it** when retrieving the movie from the database in the GetForEditByIdAsync() method in your MovieService.

A screen shot of a computer program

AI-generated content may be incorrect.

### Testing the Edit Page

A screen shot of a movie

AI-generated content may be incorrect.

What We Expect:

* The form loads with **existing movie data** **pre-filled**
* Submitting the form **saves the changes to the database**
* The app redirects back to the **Details** view of the updated movie

## Implement the Delete Operation

For educational purposes, it's very useful to teach both **soft delete** and **hard delete**, especially to understand the difference and when each is appropriate.

Why Two Approaches?

* **Soft delete** is useful when you want to allow recovery, auditing, or avoid breaking relationships
* **Hard delete** is permanent and should be used with caution

### Interface - Update IMovieService

A screen shot of a computer

AI-generated content may be incorrect.

### Service – Update MovieService

A computer screen with text

AI-generated content may be incorrect.

A computer screen with text

AI-generated content may be incorrect.

### Controller – Update MovieController

A screen shot of a computer program

AI-generated content may be incorrect.

### Movie Delete View



### Testing the Delete Page

A screen shot of a movie

AI-generated content may be incorrect.

## Workshop II – GitHub Version

### Version Available Online

* The completed solution for **Workshop II** is published and accessible at:

<https://github.com/KTsaneff/ASP.NET-Core-SoftUni-CSharpWeb-May-2025-CinemaApp/tree/Workshop-II>

## What Comes Next

* Register and log in users
* Restrict access to Add, Edit, and Delete
* Implement the **Watchlist feature**, allowing logged-in users to:
  + Add movies to their watchlist
  + View and manage their personal list